Improving Design and Implementation of OO Container-like Component Libraries

Jordi Marco and Xavier Franch
Dept. Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
c/ Jordi Girona 1-3 (Campus Nord, C6)
E-08034 Barcelona, (Catalunya, Spain)
jmarco@lsi.upc.es,franch@lsi.upc.es

ABSTRACT

Object-oriented design is usually driven by three main reusability principles: step-by-step design, design for reuse and design with reuse. However, these principles tend to be just partially applied to the subsequent object-oriented implementation, often because they conflict with other quality criteria (remarkably, efficiency). So, there is a gap between design and implementation: due to these conflicts developers use to give up design level abstractions during the implementation. In this paper we present a framework for bridging this gap for a concrete domain, the design and implementation of object-oriented container-like component libraries, such as JCF, STL, Booch Components, LEDA, etc. At the core of the framework we propose a new design pattern called *Shortcut* together with its corresponding implementation. The Shortcut pattern, introduced in a generic base class container, provides a secure and efficient access to items in a container decoupled from the implementation details of concrete containers. Shortcut enhances applying the same principles that drive the design process to the implementation process of these libraries: step-by-step implementation, implementation with reuse and implementation for reuse without penalising other quality criteria. Our framework not only supports the design and implementation of new libraries but also the reengineering of existing ones to overcome some of their drawbacks. We show by a case study, reengineering the Booch Components in Ada95, the application and benefits of our framework.

1. INTRODUCTION

Since McIlroy proposed in 1969 the notion of software catalogue [18], component-based software development (CBSD) [4, 24] has become without any doubt one of the most important software development paradigms. The key point behind this paradigm is the process of reusing components from standard software catalogues. Component reuse provides many advantages, remarkably software production hastening, software quality improvement and software maintenance cost decrease.

One of the most valuable contributions in CBSD is object-oriented (OO) technology [17, 23]. Basic features such as inheritance, polymorphism and dynamic binding, and others built on top of them, such as design patterns [13], had a strong impact on this paradigm. In fact, as Meyer points out [17, Chap. 4], OO technology makes possible, for the first time, the idea of turning academic McIlroy's vision of software development into a component-based industry.

Unfortunately, despite this ongoing success, we have not yet reached the ultimate McIlroy's goal: making software components equivalent to other engineering component. One of the reasons has to do with the difficulty of turning designs into implementations. The OO design process is usually driven by three main reusability principles [7, Sec. 2.3]:

- *Step-by-Step Design.* Design of systems as collections of structured modules which can be implemented independently.
- *Design with Reuse.* Making as reuse as possible during the (step-by-step) design of new systems.
- *Design for Reuse.* Design of components with a high potential for reuse in future applications.

However, these principles tend to be used just partially applied to the subsequent OO implementation, often because they conflict with other quality criteria (mainly efficiency). Therefore, a gap between design and implementation appears with respect to these reusability principles. This fact holds even if advanced OO features such as design patterns are used during development: it is not enough to reuse design patterns, a good reuse policy must provide implementations of them.

In this paper we present a framework [10, 14] based on the *Shortcut* notion, for bridging this gap for a concrete domain, the design and implementation of container-like component libraries (*CLC-libraries* for short). Containers (also known as collections) are holders that let us store and organise objects in different ways. Different types of containers may offer different functionalities, and their implementations (usually, more than one) will satisfy different efficiency requirements. Some representative CLC-libraries are the Java Collection Framework (JCF) [2], the Standard Template Library (STL) [22], the Library of Efficient Data types and Algorithms (LEDA) [21] and Booch Components (BC) [8, 9].

In CLC-libraries a well-established hierarchy of components can be found at the design level. Sometimes, this hierarchy

appears explicitly (e.g., JCF and BC) and sometimes not (e.g., STL and LEDA). In any case, efficiency constraints provoke that the hierarchy has no subsequent implementation in these libraries, appearing thus a gap. As usual, contradicting forces lead to different tradeoffs in the different libraries. Typically, three different situations can be found:

- The implementation of the intermediate levels of the hierarchy only provides interfaces, but not real implementations. In other words, the hierarchy implementation just preserves the layout. Code reuse does not take place, but efficiency is optimal, because common methods can use the private attributes of concrete class implementation.
- The intermediate levels simply disappear, maybe because it can be argued that non-implemented interfaces are useless from an implementation point of view. Again, efficiency is the main motivation behind this approach. Examples of this case are the STL and LEDA libraries.
- The hierarchy is partially preserved in the implementation. There are some abstract classes in the hierarchy that declare a few abstract methods and use them in default implementations of other methods. Often these methods are not as efficient as possible. JCF and BC are representative libraries in this scenario.

In any of the situations outlined above, the reusability principles become damaged:

- *Step-by-Step Implementation.* In the first two situations, each container implementation is encapsulated in an isolated module that is written in a single step, without any kind of integration in an implementation hierarchy. In the third situation, the modules of the intermediate levels of the hierarchy are not fully-implemented; in some cases, even dependencies between modules may be found.
- *Implementation with Reuse.* There is no reuse in the two first situations and not much in the last.
- *Implementation for Reuse.* There are two important properties that cause class libraries to be reusable [15]: generality (ability of one system to serve in a large range of circumstances) and extensibility (ability of a system to be easily modified to better meet a particular need). Clearly, extending the library in the two first situations requires starting from the scratch and, as we will show in Sec. 2, some solutions for the last situation (e.g., the BC solution) make assumptions about inheritor classes that provoke several problems when extending the library, and might even prevent this extension.

Our framework allows to keep the same reusability principles during the implementation process that drove the design process. This is achieved by means of a new design pattern called *Shortcut*, together with its corresponding implementation, that we propose at the core of our framework.

The Shortcut pattern encapsulates the feature of location or position of an object in a container. Shortcuts provide an abstract, reliable and efficient alternative access path to the elements stored in the container. Most of the existing CLC-libraries recognise the need for such a kind of alternative access method and thus they have similar mechanisms but they are *ad-hoc*, implementation-dependent and unreliable proposals. Instead, we provide an implementation-independent approach based on the use of shortcuts to implement a generic container which acts as a base class of the rest of containers. Shortcuts allow to implement only once, in the base class, the most common capabilities (e.g., iterators) in a highly efficient and reliable way. The implementation of both the shortcuts and the common capabilities are decoupled from the details of the implementation of its inheritors. As a result, we will show that CLC-libraries developed using our framework not only improve reusability but also other quality criteria including efficiency. Moreover, the majority of drawbacks that are present in most widespread CLC-libraries (see Sect. 2) are solved.

The Shortcut base framework not only allows the design and implementation of new libraries but also reengineering of existing ones to achieve all the benefits mentioned above. Remarkably, in this reengineering process few implementation changes are needed; the core data structures and algorithms can be reused. Since the addition of shortcuts will not affect the former observable behaviour of the departing library, the running software applications that use the previous version of the library do not require to be modified. In Sect. 5, we will show how are achieved these properties in a particular case, the Ada95 BC Library.

2. ANALYSIS OF SOME REPRESENTATIVE CLC-LIBRARIES

In this section we analyse four widespread CLC-libraries: the Java Collections Framework (JCF), the Standard Template Library (STL), the Booch Components (BC) library and the Library of Efficient Data types and Algorithms (LEDA), which are quite representative of the current state of the art in the domain. All of them differ in some characteristics. Given the goal of the paper, we focus on: depicting the general hierarchy of classes for the containers domain; identifying the most important common capabilities of these containers (e.g., type of iterators); and making explicit the drawbacks present in these libraries that we want to overcome through the Shortcut-based framework. Concerning the drawbacks, we structure their discussion by showing how the three implementations principles are not fulfilled, and paying also attention to the functionality offered by the libraries and their data integrity, which are other two issues that must be addressed given their importance. Concerning the implementation for reuse principle, we focus on the generality and extensibility properties mentioned in the introduction.

2.1 The Java Collection Framework

The Java Collections Framework (JCF), introduced in 1998 in the Java JDK 1.2, has become the standard CLC-library for Java. JCF's design goals focus on simplicity and extensibility by employing the Java language features.

JCF Hierarchy

JCF's hierarchy is based on the definition of interfaces for container classes. Its hierarchy includes two independent base interfaces: *Collection* and *Map*. Derived from these base interfaces are the interfaces of concrete containers: *Set* and *List*, which inherit the *Collection*'s interface, and *SortedMap*, which inherit the *Map*'s interface. For each of these interfaces the JCF offers an abstract implementation, which defines a few abstract methods that are used in default implementations of other methods of the container. Other interfaces named *SortedList* and *SortedMap*, which are children of *List* and *Map* respectively, are offered too. JCF offers some implementations appearing as leafs of the hierarchy, and a hierarchy of iterators. Figure 1 shows the complete hierarchy of JCF containers as presented in [2].

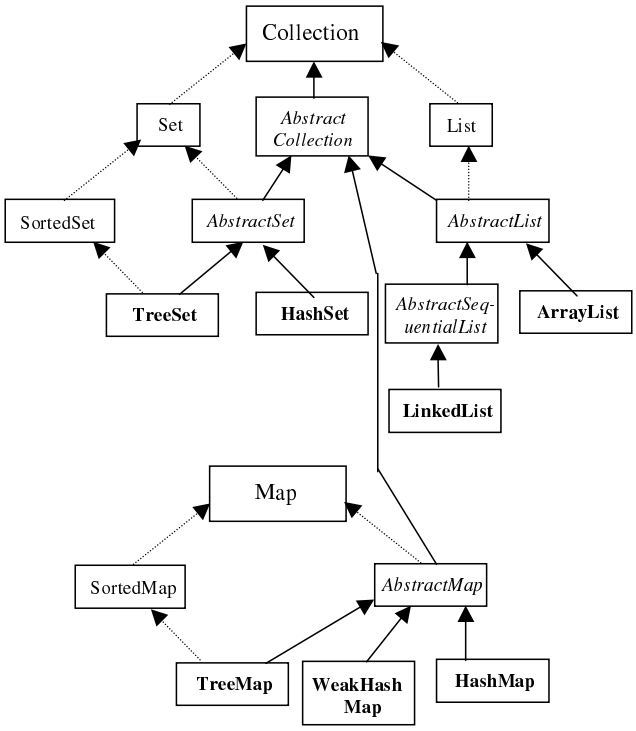


Figure 1: The JCF hierarchy of collections.

JCF Common Capabilities

- Iterators. The JCF common iterators are forward ones. Only *List* objects provide bidirectional iterators (*ListIterator* in JCF).

- Insert operations. All the JCF collections offer an *add* operation that inserts an item into them. In addition, maps offer a *put* operation that inserts pairs of key and value into the map; this approach is different from the usual one that force map items to be decomposed into key and value.

- Deletion, modification and access operations. JCF only provides common deletion operations. These common operations are offered by the *Collection* and *Iterator* interfaces.

- Locations. Although JCF does not offer this capability, the *reference* feature of Java can be used for this purpose.

JCF drawbacks

- Step-by-step implementation. There are a few implementation steps, because intermediate nodes of the library are abstract implementations with not much code in them. Only leafs of the hierarchy are full implementations.

- Implementation with reuse. Some common capabilities (remarkably iterators) are implemented from the scratch on each abstract implementation of the containers' hierarchy and in some cases leaf classes override the methods implemented in abstract classes due to efficiency requirements (e.g., *HashMap* override *get* and other key-based methods).

- Implementation for reuse: generality. JCF offers a few different containers, which may not be appropriated for a concrete scenario. This is due to the fact that JCF is a framework, not a library, then it only provides the core set of classes for containers. The extension of this set which other containers more adequate to a specific scenario is left to the JCF user.

- Implementation for reuse: extendibility. JCF abstract implementations are designed to be easily extended, but efficiency is not always a consequence.

- Functionality. Common iterators are just forward and cannot be combined arbitrarily with updates. The concept of location for direct access to elements in the container does not exist.

- Integrity. Iterators may become out-of-date in some container classes when a new object is inserted to, or an existing one is removed from, the container. JCF containers do not offer operations to know if an iterator is still valid or not. The usage of the *reference* Java feature as location provokes out-of-date access if the object is not in the container.

2.2 The Standard Template Library

STL is a component library adopted by ANSI/ISO as a standard for C++ [3]. STL has not an explicit class hierarchy; however, there is an implicit hierarchy that can be deduced from its internal structure.

STL (implicit) Hierarchy

STL containers are divided into two categories: sorted associative containers and sequence containers. The associative containers category offers four different containers abstractions: sets, multisets, maps and multimaps, each of them with just one single implementation.

The scheme is slightly more complicated in the case of sequence containers. There are three different sequence containers abstractions: vectors, deques and lists, all of them with a single implementation. In top of them, stacks, queues and priority queues are defined. Therefore, it can be said that STL has six different sequence containers abstractions: vectors, deques and lists have just an implementation, while

stacks, queues and priority queues have three different implementations each one.

STL Common Capabilities

- Iterators. STL iterators are divided into five categories: input iterators, ouput iterators, forward iterators, bidirectional iterators and random access iterators among which there is the hierarchical relationship shown in Fig. 2. Only vectors and deques provide random access iterators; the other containers provide just bidirectional ones.

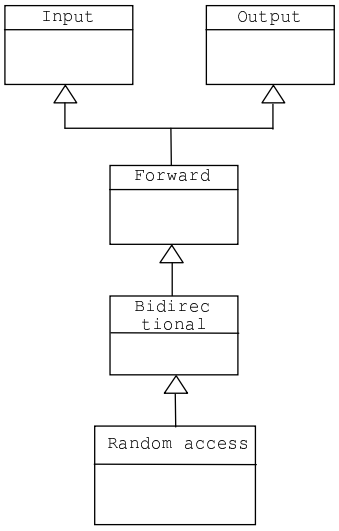


Figure 2: The STL iterators hierarchy.

- Insert operations. There are a variety of insert operations to insert one or more objects. The common one is the basic insert operation which consists in inserting one object; this operation returns an iterator that can be used like a reference to the new object.
- Deletion, modification and access operations. All of them use iterators as pointers to manipulate the object they are bound to.
- Locations. In STL, iterators mix two different capabilities, namely, the concept of iterator and the concept of location of the objects stored in the container for efficient individual access.
- Generic algorithms. STL offers several generic algorithms that work over all containers trough iterators.

STL drawbacks

- Step-by-step implementation. STL containers are implemented in only one step.
- Implementation with reuse. Some common capabilities (remarkably iterators) must be implemented from the scratch on each concrete class.
- Implementation for reuse: generality[1]. Container abstractions have very few implementations available (most of them just one). Although underlying data structures are quite appropriate, some contexts demand implementations with requirements that cannot be entirely satisfied with the existing ones (e.g., hashing for associative mapping containers instead of the provided red-black tree, remarkably when perfect hashing could be used). This drawback could be solved extending the library but it is difficult in STL.
- Implementation for reuse: extendibility. Extending STL with a new container requires starting from the scratch. Moreover, some of the common capabilities that must be implemented restrict the set of candidate implementations (e.g., internal rearrangement of elements when updatings take place are not allowed due to the iteration capability). Other capabilities are difficult to understand and/or implement (e.g., iterators with pointer arithmetic).
- Functionality. Updating the container is not allowed during iteration, since iterators may become out-of-date due to internal rearrangements of elements (remarkably in array-based implementations).
- Integrity. Iterators may become out-of-date in some container classes when a new object is inserted to, or an existing one is removed from, the container. Even in the rest of container classes, an iterator may become out-of-date if the object that it refers to has been removed using the operations of the container. STL containers do not offer operations to know if an iterator is still valid or not.

[1] We mean generality with respect to containers, not to algorithms.

2.3 Booch Components

The Booch Components (BC) Library was first created for Ada 83 [5], reengineered for C++ [8] and for Ada 95 [9]. We analyse in detail the newest one, the Ada 95 version, which is later used in the paper to show the feasibility of our approach (see Sect. 5).

BC Hierarchy

The *Containers* category of classes in the BC Library provides a wide range of structural abstractions (lists, bags, sets, collections, etc.). Each container is implemented following a single strategy (chaining, hashing, search trees and so on), using three different implementation classes (*Bounded*, *Unbounded* and *Dynamic*) which differ in minor details concerning the management of the size of the container. Figure 3 shows the main level of the hierarchy containing most of its structural abstractions; each of them offers the interface of its operations. The subhierarchy of one of the abstractions is also depicted, the implementation classes appearing as leaves in the tree.

BC Common Capabilities

- Iterators. BC provides only forward iterators. The *Containers* class offers the interface of them, while the structural abstraction classes offer the implementation of iterators. The intended goal is that all the implementation classes of an abstraction share its implementation of containers supporting thus code reuse; this intent provokes serious drawbacks due to some dependencies between parent and children cases (explained in more detail in Sect. 5).

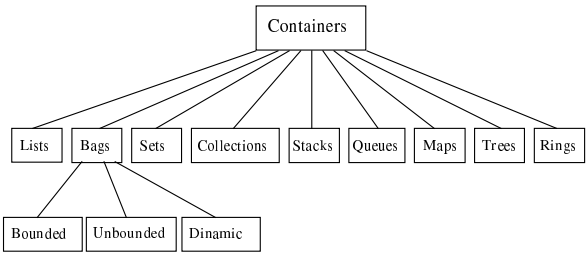


Figure 3: The BC hierarchy of containers (excerpt)

- Insertion operations. All the containers offer an *Add* common operation that inserts an item into the container.

- Deletion, modification and access operations. There are no common operations of these kinds; each structural abstraction offers its specific operations.

BC drawbacks

- Step-by-step implementation. BC containers are implemented in incomplete and not independent steps (implementation parts of some parent-classes depend on the concrete implementation of their children classes).

- Implementation with reuse. The iterator type and its operations are strongly dependent on the concrete implementation strategy of the underlying structure. As a consequence, for every concrete implementation of a children-class, a new iterator type must be defined and its operations must be overridden.

- Implementation for reuse: generality. The design of the hierarchy supports just a single implementation strategy (although with three different memory management schemes) for every different structural abstraction. This is a serious drawback because the chosen implementations can be efficient in some contexts but not efficient enough in others (e.g., hashing vs. red-black trees).

- Implementation for reuse: extendibility. Structural abstraction classes make assumptions about the implementation of their inheritor classes. These assumptions provoke extensions of the class hierarchy with different implementations, or changes in existing ones, to be not always possible. Even when changes are possible, changing an implementation may require changing some of its parents (see Sect. 5). Also, implementation changes may affect iterators' code because they are implementation-dependent.

- Functionality. Iterators are just forward and cannot be combined arbitrarily with updates. The concept of location for direct access to elements in the container does not exist.

- Integrity. Iterators may become out-of-date if insertions and deletions are made while the container is being traversed.

2.4 The Library of Efficient Data types and Algorithms

The Library of Efficient Data types and Algorithms (LEDA) is the result of a project started in 1988. The actual version of this library [21] offers a large amount of data types and algorithms, mainly to be used in the combinatorial computing area, being in this respect more powerful than the majority of other container libraries.

LEDA (implicit) Hierarchy

Although there are some common capabilities in LEDA container-like data types, there is no an explicit hierarchy of classes. Even in the case of different implementations of the same containers, LEDA uses parameterisation instead of inheritance. Thus, many of the LEDA containers-like data types (dictionaries, priority queues, sorted sequences, etc.) require an additional data structure parameter for choosing a particular implementation.

LEDA Common Capabilities

- Iterators. LEDA offers iteration macros that can be used similarly to the C++ **for** statement to iterate through the objects of a container in both directions.

- Insertion operations. The different existing insertion operations return an item type that can be used in other operations which have then constant time.

- Deletion, modification and access operations. These operations take items as parameters to manipulate the object they refer to. Their precondition states that objects bound to these items must already be in the container; so, any assumption about the behaviour of these operations when the involved objects are not in the container may fail.

- Locations. LEDA offers a item type that plays the role of location or position in data structures, to gain efficient access to the objects therein.

LEDA drawbacks

- Step-by-step implementation. LEDA containers are implemented in only one step.

- Implementation with reuse. As far as no OO hierarchy is explicitly provided, all the common capabilities have to be implemented on each concrete class (remarkably, the item concept and the iteration macros).

- Implementation for reuse: generality. Some assumptions about container elements functionality imply that containers cannot be used in some circumstances (e.g., they must provide an operation for writing their value).

- Implementation for reuse: extensibility. It is difficult to extend it. Some of the common capabilities that must be implemented restrict the set of candidate implementations (e.g., containers whose objects change their position in the internal structure when a new object is inserted would lead to inconsistent states). Other capabilities are not obvious to understand and/or implement (e.g., items used sometimes as pointers).

- Functionality. Iteration macros have the restriction that the object corresponding to the item should not be modified inside the body of the loop; furthermore, insertions are not allowed as well as deletions of elements others than the current one.

- Integrity. An item could become out-of-date if the object that it referred to has been removed. There is not an operation to know if an item is still in a container or not. Although iteration and updating cannot be combined arbitrarily, this is not explicitly controlled and it becomes thus responsibility of the user.

2.5 Summary of results

The analysis carried out in this section shows that none of these libraries satisfy all the reusability principles altogether. More precisely, there is no *Step-by-step Implementation* (only one step, incomplete steps or non-independent steps), low or absent *Implementation with Reuse*, and the *Implementation for Reuse* principle is seriously damaged, mainly regarding extensibility: their maintenance is difficult and they may not allow arbitrary enlargement. Also, other common drawbacks appear: iterators and access by location are not entirely secure due to the improper management of the interference between "usual" operations (i.e., those bound to the abstraction itself, as deletion for key) and those involving iterators and locations. Furthermore, each library suffers from its particular drawbacks, for instance utterly unsafe use of iterators in STL and absence of access by location in the BC Library. Then, we conclude that a framework offering all the common capabilities and solving these drawbacks simultaneously is quite desirable.

3. THE SHORTCUT DESIGN PATTERN

Our goal is to define a framework for bridging the gap between design and implementation in the domain of CLC-libraries, with respect to the three reusability principles. Also, we want to overcome the other different drawbacks presented in the previous section. To achieve this goal we propose a new design pattern that we will use at the core of our framework. We present next this pattern: the *Shortcut* design pattern, using the format presented in [13].

Intent

Define an object that encapsulates the concept of location or position of an object in a container. In other words, provide an abstract, efficient and reliable way to access to the objects in a container without exposing its underlying implementation. The design requirements we consider are:

- Time efficiency. In particular, access to the elements stored in a container by means of their shortcut must be achieved in constant time $O(1)$ [6].

- Integrity. In particular, meaningless access by shortcuts must be detected and avoided.

- Robustness. We say that a location or position is robust if:

  - It is bound to one and only one object in the container.
  - It does not change while the object which it is bound to is inside the container, even if the underlying representation requires rearrangements.
  - It is possible to know if it is bound to an object in the container or not.

- Evolvability. Shortcuts are created and destroyed as the elements are inserted in, and deleted from, the container.

Also Known As

Location or position

Motivation

As we will show in Sect. 4, this pattern allows bridging the gap between design and implementation reusability principles in the domain of CLC-libraries. As a consequence, it solves the problems due to this gap that damage CLC-libraries. Moreover, this pattern provides fast access to the objects stored in a container in those contexts where efficiency is crucial. And finally, it allows maintaining robust references to objects stored in a container, in other objects, which is a common practice in programming with CLC-libraries.

Applicability

Use the *Shortcut* pattern to fulfil the following requirements altogether:

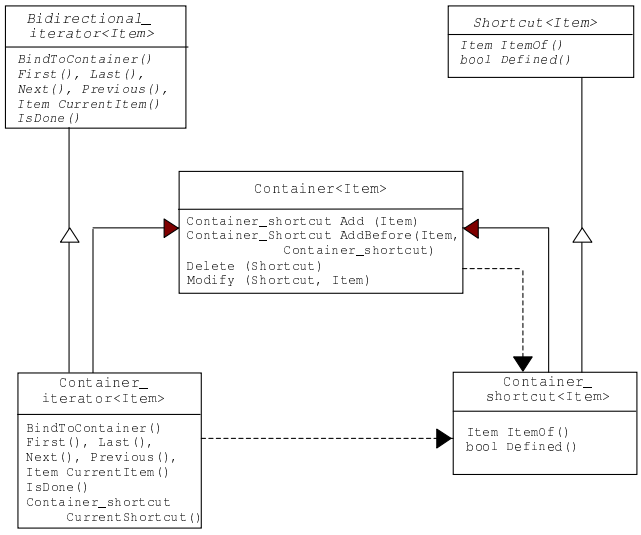
- To access to the objects stored in a container in an abstract and efficient way.

- To maintain robust references to objects stored in a container, in other objects (as we will show in Sect. 4 this objects can be concrete containers).

- To offer a uniform interface to perform efficient operations in a container.

- To support entirely the application of the three reusability principles on design and implementation of CLC-libraries.

Structure



Participants

- **Bidirectional_iterator**
  - Defines the interface of bidirectional iterators, i.e. iterators that support forward and backward traversals of a container. We assume that this class is part of a complete hierarchy of iterators[2], like the one of STL, which includes input, output, forward, bidirectional and random iterators.
- **Container_iterator**
  - Provides an efficient implementation of Bidirectional_iterator enlarged with a new method that returns the shortcut bound to the current item of the iterator. All the operations of this class must be $O(1)$.
  - Keeps track of the current position in the traversal of the container.
- **Shortcut**
  - Defines an interface for accessing the elements in an abstract and reliable way. More precisely, the operations are: *Defined*, that checks whether a shortcut is valid or not; and *ItemOf*, that gets the object bound to a valid shortcut.
- **Container_shortcut**
  - Implements in an efficient manner the *Shortcut* interface.
- **Container**
  - Defines the interface of those general-purpose container operations that involve shortcuts, i.e. *Add*, *AddBefore*, *Delete* and *Modify*, as well as *Nitems* to find out the number of items in the container.
  - Implements in an efficient and safe manner the *Add* operation in a way that returns a shortcut to the new object. The object is added in the last position in the iteration ordering.
  - Implements in an efficient and safe manner the *AddBefore* operation to add an object before (in the iteration ordering) the object bound to a shortcut. This operation is useful in contexts requiring iterations in orderings other than the implicit add-ordering (e.g., ordering by contents).
  - Implements in an efficient and safe manner the *Delete* and *Modify* operations by means of their shortcut.

Collaborations

- `Container_shortcut` keeps track if an object is still in the container and if it is the case the shortcut allows accessing to it.
- `Container` is responsible of notifying deletion of objects to the corresponding shortcuts.

[2]This hierarchy has not been depicted in the previous Structure section for space-saving purposes.

Consequences

The Shortcut pattern offers several benefits to container's inheritors:

1. The objects stored in a container can be accessed without knowing how they are stored therein and, therefore, without knowing the underlying representation of the container (with arrays, pointers, linked, in tree-form, ...).
2. The access to the objects in a container by means of a shortcut is done in O(1) time (see Implementation section below), making it possible to reuse containers even in those contexts with high efficiency constraints. Moreover, the efficiency of the iterator operations is $O(1)$ in the worst case[3].
3. The access to the objects by means of shortcuts is safe because meaningless access to them is avoided. In particular, the following situations are avoided: dangling shortcuts (references without bound objects) and out-of-date ones (shortcuts bound to objects which are not the original ones).
4. The children classes of *Container* inherit its definition of shortcut operations. Therefore, the objects stored in concrete container can be accessed by using either the operations that characterise every particular type of container or the shortcut operations.
5. Iterations are not comitted to a concrete container implementation. Iterations are made over the Container base class and as a consequence we can iterate over a container without committing to an specific implementation[4].

Implementation

The essential point consists in maintaining an efficient mapping from shortcuts to items in the *Container* base class. There are three possibilities to implement this mapping depending on the underlying memory management scheme:

1. *Using dynamic storage without garbage collection.* In this case the *Shortcut* class is implemented with a *smart pointer* [11]. Smart pointers are characterized by the fact that deletion of allocated objects does not take place until there are no shortcuts bound to them. The smart pointer points to a tuple containing the object and a *deleted* flag, i.e., an attribute to record if the object is deleted or not. On the other hand, the *Container* base class is implemented with a double linked list of these tuples in order to have efficient bidirectional iterators.
2. *Using dynamic storage with garbage collection.* The smart pointer can be sustituted by a regular pointer. There are no more changes with respect the previous approach.

[3]We would like to remark that this low cost cannot be assured with implementation-dependent iterators, if these implementation is not well-suited for this purpose (e.g., a hashing implementation).

[4]Polymorphic iterators solve this problem too, but they introduce other drawbacks.

3. *Using an array.* In this case shortcuts are implemented as indexes to the array position. Then the *Container* base class is implemented as an array of tuples which contain the object, a deleted flag, a reference counter (being every reference a shortcut) and two indexes to the next and previous tuples in the iterator ordering. As released shortcuts (the deleted array positions) must be available somehow to allow further reassignment, we must link them too. Additional index members corresponding to the position of the first object, the position of the last object and the first free position have to be maintained in the *Container* base class.

It must be remarked that shortcut assignment is properly managed in both schemes. Using dynamic storage, the class defining smart pointers redefines the assignment to detect that a new shortcut has come up. Using an array, a reference counter is explicitly added and assignment is redefined in the same way.

It can be observed that shortcuts require some extra space in order to get all its benefits. More precisely, being N the number of objects in the container and assuming a pointer-based implementation of shortcuts without garbage collection, which is the worst case concerning space efficiency, the total amount of extra space is:

$$2 \cdot N \cdot space(\text{pointer}) + N \cdot space(\text{shortcut}) + N \cdot space(\text{bool}) + N \cdot space(\text{integer})$$

The first operand comes from the double linked list[5], the second one from the shortcuts stored in the implementation of a concrete abstraction and the last two operands from the deleted flag and the reference counter. However, this waste of space will usually generate a later saving, when shortcuts substitute identifiers (generally strings, which require more space than shortcuts) in references from other objects. The relationship between these two factors may be formally established. Let N be the total number of objects in the container and R the total number of references to objects in the container. Since generally,

$$space(\text{identifier}) \geq space(\text{pointer}) \quad (1)$$

then $\exists k \geq 1$ s.t. $(k+1) \cdot space(\text{pointer}) > space(\text{identifier}) \geq k \cdot space(\text{pointer})$ and since $space(\text{shortcut}) = 2 \cdot space(\text{pointer})$ (because we use two pointers for assuring that a particular shortcut is bound to a particular container) and assuming the worst case $space(\text{bool}) = space(\text{integer}) = space(\text{pointer})$, space is really saved when the relationship

$$R \cdot space(\text{identifier}) \geq (6 \cdot N + 2 \cdot R) \cdot space(\text{pointer}) \quad (2)$$

holds, which is satisfied when the following condition holds:

$$R \cdot k \geq 6 \cdot N + 2 \cdot R \quad (3)$$

For example, if identifiers were strings of average size 40 (i.e., $k = 10$ with pointers requiring 4 bytes) and there is an external reference to each of them ($N = R$), then by substituting from (3) we obtain:

$$10 \cdot R \geq 8 \cdot R$$

that obviously holds.

[5]The space of a smart pointer is the same as a regular one.

Sample code

We outline here a C++ implementation of three participants in the proposed structure, *Container_shortcut*, *Container_iterator* and *Container*. The complete code can be found in [20]. The implementation of shortcuts is made using dynamic memory.

1. *Implementation of the Container_shortcut class.* In the implementation of this we use: a **pointer** class, which encapsulates the definition of smart pointers and a **Node** class, which contains the item and the deleted flag.

```
template<class Item>
class Container_shortcut {
  protected:
    pointer<Node> ptr;
    Container* ptrContainer;
    friend class Container<Item>;
    friend class Container_iterator<Item>;
  public:
    Container_shortcut() :ptrContainer(NULL) {}
    Container_shortcut(const Container_shortcut&
         shortcut):ptr(shortcut.ptr),
         ptrContainer(shortcut.ptrContainer) {}
    const Item& ItemOf() {
        if (!Defined()) throw UndefinedShortcut;
        return ptr.value().item;
    }
    bool Defined() {
        if (ptr.IsNull()) return false;
        if(ptr.value().deleted_flag) {
          ptr.SetNull(); return false;}
        return  true;
    }
    void operator =(const Container_shortcut&
                    shortcut) {
        ptr = shortcut.ptr;
        ptrContainer = shortcut.ptrContainer;
    }
    ~Container_shortcut(){}
  protected:
    Container_shortcut (const Item& item) {
        ptr = pointer<Node>(Node());
        ptr.value().item = item;
    }
};
```

The implementation of this class stores a smart pointer **ptr** to the container node where the item bound to the shortcut is, along with a pointer to the **Container**. Notice that the operations **ItemOf** and **Defined** are not declared as **const**; if the object associated to the shortcut is marked as deleted, the pointer is set to **NULL** to dismiss the number of references. The assignment of shortcuts is defined in top of the assignments of smart pointers. All the operations need $O(1)$ time in the worst case.

2. *Implementation of the Container_iterator class.* We present here an excerpt of its implementation

```
templete<class Item>
class Container_iterator
:public bidirectional_iterator<Item> {
  protected:
   pointer<Node> ptr;
   Container<Item>* ptrContainer;
  public:
   Container_iterator() :ptrContainer(NULL) {}
   Container_iterator(const Container_iterator&
       iterator):ptr(iterator.ptr),
       ptrContainer(iterator.ptrContainer) {}
   ~Container_iterator(){}
   virtual void BindToContainer(Container<Item>* C)
   {// current element set to first element
       ptr = C->_container.value().next;
       ptrContainer = C;
```

```
    }
    virtual void First() {
     if(ptrContainer != NULL)
        ptr = ptrContainer->_container.value().next;
    }
    virtual void Next() {
       if(!IsDone()) ptr = ptr.value().next;
    }
    virtual const Item& CurrentItem() {
        if(IsDone()) throw IteratorIsDone;
        return ptr.value().item;
    }
    ...  // other operations
    virtual Container_shortcut<Item>
        CurrentShortcut() {
      Container_shortcut<Item> sh;
      if(IsDone()) throw IteratorIsDone;
      sh.ptr = ptr; sh.ptrContainer = ptrContainer;
      return sh;
    }
};
```

This class offers the new operation **CurrentShortcut** that returns the Shortcut associated to the **CurrentItem**. Smart pointer assignment inside this method keeps track of the existence of a new shortcut to the involved object (this also happens in iterator assignment); also, the container bound to the shortcut is the container bound to the iterator through **BindToContainer**. The rest of operations are the usual ones for iterators and are implemented straightforwardly. As in the case of the **Shortcut** class, all the operations need O(1) time in the worst case.

3. *Implementation of the class Container.* To get the flavor, we present just the **Add** and the **Delete** (by shortcut) operations. **Add** implements the insertion in a double linked list while **Delete** implements the deletion marking as deleted the item associated to the shortcut, if any, setting to **NULL** the shortcut members **ptr** and **ptrContainer**, and the **Node** pointer members **next** and **previous**. We remark that the time efficiency is optimal, O(1), as it was required[6].

```
template<class Item>
class Container {
  protected:
    friend class Container_iterator;
    friend class Container_shortcut;
    pointer<Node> _container;
    unsigned long count;
  public:
    ...
    virtual Container_shortcut<Item> Add(const Item
                                         &item);
    virtual void Delete(Container_Shortcut<Item>&
                        shortcut);
    virtual void Modify(Container\_shortcut<Item>&
                        shortcut,const Item& item);
};


template <class Item>
Container_shortcut<Item> Container<Item>
::Add(const Item &item) {
  Container_shortcut<Item> sh(item);
  _container.value().previous.value().next=sh.ptr;
  sh.ptr.value().previous=
                      _container.value().previous;
  sh.ptr.value().next=_container;
  _container.value().previous = sh.ptr;
  count++; sh.ptrContainer = this;
  return sh;
}


template <class Item>
void Container<Item>
::Delete(Container_shortcut<Item> &shortcut) {
  if (shortcut.Defined()) {
    if(shortcut.ptrContainer != this)
      throw ShortcutIsNotBoundToThisContainer;
    shortcut.ptr.value().deleted_flag = true;
    shortcut.ptr.value().previous.value().next =
                      shortcut.ptr.value().next;
    shortcut.ptr.value().next.value().previous =
                  shortcut.ptr.value().previous;
    shortcut.ptr.value().next.SetNull();
    shortcut.ptr.value().previous.SetNull();
    shortcut.ptr.SetNull();
    count--;
  }
}
```

[6] Section 5 presents some accurate computational results.

Known Uses

Shortcuts are widely used to keep track of references to objects stored in a container, in other objects. This is a usual functionality in the most widespread CLC-libraries. A typical example is to maintain in an object that is in a container a reference to other object in the same container (e.g., a parent-children relationship in a list of people); if shortcuts are not used, we either get non-efficient and unsafe access to elements or lose code reuse.

Another common use is to combine different data structures in an efficient manner to obtain new efficient ones. An example is the case of a symbol table in the program compilation field. Although a symbol table can be described and understood in terms of classical containers (stacks, lists and maps), normally they are not really reused; instead, the data structure is represented directly in terms of arrays and pointers. Integrity problems can cause then very serious problems. We can find an example in [25].

4. A SHORTCUT-BASED FRAMEWORK FOR CLC-LIBRARIES

In this section we define the *Shorcut-based Framework* (SBF), based on the *Shortcut* pattern presented in the previous section. The key point for reaching our goal (bridging the gap between design and implementation) consists on storing the objets of any concrete container in the **Container** class of the Shortcut pattern, while keeping in the concrete container only the shortcuts bound to them. We show throughout this section the complete development of this idea. We first define the Shortcut-based Framework hierarchy for CLC-libraries. This hierarchy has been built borrowing the main ideas from the different hierarchies of the CLC-libraries studied in Sect. 2. In fact, we are not interested in fixing all the details of the hierarchy (i.e., which concrete containers, and which concrete operations in them, do exist), but its general layout. Next, we will implement the parts of the framework that are not included in the Shortcut pattern. Last, we will outline the benefits coming from the existence of the framework.

4.1 Hierarchy of SBF

From the analysis carried out in Sect. 2 it can be deduced that a complete hierarchy of a CLC-Library must provide: a container base class (where common capabilities are offered); a hierarchy of iterators; locations for accessing and modifying containers; concrete containers classes; and different implementation strategies for each concrete container. We have introduced in the previous section the Shortcut pattern which covers the common elements (the base class, the iterator hierarchy and the locations, i.e., the shortcuts). Now we

integrate the concrete containers and their implementations to build the framework.

Our objective is to reuse in the concrete containers classes the common capabilities of the (fully-implemented) container base class. In the OO methodology there are two common techniques for reusing: inheritance and composition. The basic difference between them is that inheritance provides what is know as *white-box* reuse (i.e., the features of the parent classes are often visible to subclasses) while composition provides *black-box* reuse. We use inheritance instead of composition in our hierarchy because we want to allow the definition of generic algorithms that work over any type of container (taking a container base as a parameter), but in fact we use inheritance as a black box, as shown in the next subsection.

Figure 4 shows the hierarchy we have chosen for the Shortcut-based Framework.

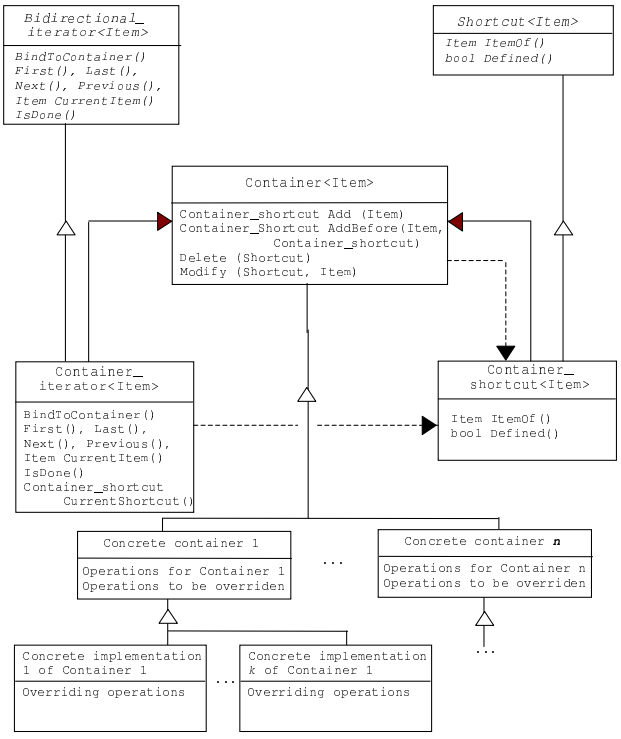


Figure 4: Hierarchy of the Shortcut-based Framework

The classes involved in this hierarchy are:

- The participants on the shortcut pattern, described in the previous section. Shortly:
  - *Bidirectional_iterator*. An abstract class that provides the interface of this kind of iterator, i.e. iterators that support forward and backward traversal of a container.
  - Container_iterator. An efficient implementation of Bidirectional_iterator enlarged with a new method that returns the shortcut bound to the current item of the iterator. This class is implemented over the base class Container; as a consequence, it is fully independent of the specific kind of container. Although the best case is having random iterators, they are difficult to implement; thus, concrete containers will commonly offer bidirectional iterators. All operations of this class must be $O(1)$.
  - *Shortcut*. Defines the interface of the new design pattern that introduces the concept of shortcut as the location or position of objects.
  - Container_shortcut. An efficient (i.e., $O(1)$ time) and safe implementation of the Shortcut interface. Container_shortcut is implemented over the base class Container; as a consequence, it is fully independent of the specific kind of container but can be used for access to items them store.
  - Container. This base class acts as a common parent class for all kinds of containers. It provides the interface and implementation of the most common capabilities of CLC-libraries:
- **`Concrete containers`**. Children classes of Container that are not leaves, which represent different types of containers (list, map, etc.). Each of them adds the interface and implementation of its specific functionalities to the ones inherited from the Container class. This interface may include specific shortcut operations (e.g., obtaining the number of repetitions of an existing object using its shortcut).

  The strategy chosen to implement these classes consists on storing the items in the base class Container and the shortcuts bound to them in a concrete implementation (an array, dynamic storage, ...). In order to do this, specific operations of concrete containers are implemented using (if it is necessary) an operation implemented by their subclasses (i.e., using the Template Method design pattern). Concrete containers also define as protected the interface of the deferred operations that appear as a result of applying the Template Method design pattern (that we call *concrete interface*) and implement a (in some cases non-efficient) version of them using the Container interface and shortcuts. We want to remark that this implementation strategy uses the base class Container as a black box and, at the same time, makes the concrete container a black box for its children classes. Moreover, all the operations of the container class are $O(1)$. Last but not least, each concrete container is an implementation (non-abstract) class.

  Unlike other possibilities (e.g., considering the implementation as a parameter or using the Strategy design pattern) we have chosen inheritance and application of the Template Method design pattern for implementing the deferred operations, to allow the concrete implementations offering new functionalities (e.g., random iterators).
- Concrete implementations. Children classes of concrete containers that are leaves. These classes implement the concrete interface. They inherit all the functionalities of the concrete container and as a consequence their implementation can be made avoiding

iterators and locations. On the other hand, inherited implementations may remain if they already fulfil efficiency requirements.

4.2 Example of using the SBF

We outline here the use of SBF for implementing a concrete container **Map** by means of a concrete implementation **MapArray**. We have chosen this example because it shows the main points of the SBF:

- The application of the Template Method design pattern.
- The use of parent and children classes as black boxes.
- The persistence of iterators and shortcuts, even when the concrete implementation makes rearrangements of items inside the underlying data structure.
- The possibility of defining generic algorithms that work over any kind of containers.

Concrete Container **Map**
This class is in the second level of the SBF class hierarchy. The concrete container **Map** bounds values to keys. We show here the interface of this class and the implementation of the map operation **Delete** (by key) which illustrates the use of the Template Method design pattern. The identifiers of the deferred operations that appear, as well as the type representations, have the string **Con** as prefix. The **ConDelete** operation deletes the shortcut stored in a concrete implementation of the container abstraction; this shortcut is returned as result and used then to delete the item. Actually this implementation of **Delete** does not depend on the concrete implementation of the map container. We want to remark that its execution time is the same as the execution time of the concrete deferred operation (shortcut operations take $O(1)$ time).

```
template <class Key,class Value,class Less=less<Key> >
class Map:public Container< KeyValue<Key,Value> > {
  ...
  protected:
   typedef Key ConKey;
   typedef Container_shortcut<Item> ConValue;
   typedef KeyValue<Container_shortcut<Item>,ConValue>
                    ConItem;
   ...
   Container_iterator<Item> cache;
  public:
   typedef KeyValue<Key,Value> Item;
   Map():conless(less) {cache.BindToContainer(this);}
   ~Map(){}
   Shortcut Add(const Item& item);
   void Delete(Container_shortcut<Item>& shortcut);
   void Modify(Container_shortcut<Item>& shortcut,
               const Item& item);
   Value Delete(const Key& key);
   const Value& Get(const Key& key);
   bool Exist(const Key& key);
  protected:
   virtual void ConAdd(const ConItem& item){}
   virtual ConValue ConDelete(const ConKey& key);
   virtual ConValue ConGet(const ConKey& key);
   virtual bool ConExist(const ConKey& key);
};
```

```
template <class Key,class Value,class Less=less<Key> >
Value Map<Key,Value,Less>::Delete(const Key& key) {
    Shortcut_container<Item> sh; Value v;
    sh = ConDelete(key);
    v = sh.ItemOf().value;
    Container<Item>::Delete(sh);
    return v;
}
```

Concrete Implementation **MapArray**
This class is at the bottom level of the SBF class hierarchy. This array-based **MapArray** implementation includes the code for the deferred concrete operation **ConDelete**. It is worth to remark that this implementation changes positions of elements in the array; in despite of these rearrangements, all the existing shortcuts and iterators keep being valid:

```
template <class Key, class Value, class Less=less<Key> >
MapArray<Key,Value,Less>
 ::ConValue MapArray<Key,Value,Less>
 ::ConDelete(const ConKey& key) {
    ConValue v; int i;
    if(!ConExist(key)) throw NotExistingKey;
    v = mapArray[cache].value; pos--;
    for(i = cache--; i < pos; i++)
        mapArray[i] = mapArray[i+1];
    return v;
}
```

A generic algorithm

We define a generic function for printing the collection of objects of a container that works for any kind of container (because iterators are independent of concrete container).

```
template <class Item>
void print(Container<Item> *C) {
  Container_iterator<Item> it;
  cout << "Printing the " << C->NItems()
       << " items in container:";
  it.BindToContainer(C);
  it.First();
  while(!it.IsDone()) {
    cout<< it.CurrentItem()<<' '; it.Next();}
  cout << ":Last\n";
}
```

The next piece of code show its use applied to the *MapArray* implementation of a *Map* container.

```
MapArray<int,int> C;
...       // code to insert elements
print(&C);
```

4.3 SBF benefits

We outline here the main benefits of applying our framework in the design and implementation of CLC-Libraries.

- Step-by-step implementation. Each SBF class implements all the common capabilities that the class offers (those ones identified in the current step), so it is an implementation class. The full implementation of each class is carried out using its parent class (implemented in previous steps) as a black box (i.e., using only its interface) and without making any assumption about the implementation of the classes appearing in the next step.
- Implementation with reuse. The most common capabilities (remarkably iterators and shortcuts) are implemented only once in the most efficient way (all the operations of shortcuts and iterators are $O(1)$).

- Implementation for reuse: generality. The generality of a Shortcut-based CLC-Library depends on the number of concrete containers and concrete implementations it offers. As far as the implementation of the hierarchy does not make any assumption about the form or behaviour of its components, the library may contain an utterly rich variety of containers and implementations. Last, a lot of generic algorithms working over the container base interface or over the container iterators can be offered.

- Implementation for reuse: extendibility. Due to the extendibility characteristic of the framework, as many concrete containers as needed may be added; in addition, there is no restriction on the number and variety of container implementations that can appear, and new implementations may be added to existing concrete containers as required. These extensions are easy to carry out because they can reuse the most common capabilities (iterators and shortcuts); furthermore, as a consequence, the implementation of these classes is not restricted for the efficiency requirements of these capabilities.

- Functionality. In addition to offering the common features of the CLC-libraries studied in section 2, updating the container during iteration is allowed (only deletion of the current item during iteration is forbidden, even in this case the unique consequence is that the iteration is finished[7]).

- Integrity. Iterators not become out-of-date when a new object is inserted to, or an existing one is removed from, the container. The SBF offer operations to know if an iterator is still valid or not and if a shortcut is bound to an item in the container or not.

5. A CASE STUDY: REENGINEERING THE BC LIBRARY

In this section we summarise the main results of applying our framework to the OO Ada 95 version (which is the most recent one) of the BC Library presented in Sect. 2. We have chosen this library because it exhibits a very satisfactory functionality, it is public, it is well-documented and there are exhaustive test cases for it; to sum up, it is one of the most widespread CLC-libraries especially in the case of Ada ones. However, we have seen in our comparative study (see Sect. 2) that it has many serious drawbacks concerning the OO implementation principles and other criteria; these drawbacks make our reengineering process particularly challenging. In [19] appears an extended version of this reengineering process from a different (more programming-language-oriented) perspective, although sections 5.3 and remarkably 5.5 are presented here for the first time. We will focus on a concrete type of container with the hierarchy stemming from it, the Bags Containers subhierarchy (see Fig. 3).

5.1 Class dependencies

[7] In fact, also this situation could be supported with a few extra cost, but we have considered it unnecessary; anyway, the deletion of the current item can be currently done using just an auxiliary shortcut during iteration

We have shown in Sect. 2 that the container category of this library presents some drawbacks that decrease its quality. Most of the problems arise because some parent-classes depend on the concrete implementation of their children-classes.

To make it clear, Fig. 5 shows a typical situation in this library, in which the *BC.Containers.Bags* class depends on the concrete form of its children classes *Bounded*, *Unbounded* and *Dynamic*, which are different forms of hashing tables. Notice that the type definition of *Bag_Iterator* in *BC.Containers.Bags* class forces all its children to be implemented by means of a hashing table.

```
generic
package BC.Containers.Bags is
 ...
private

type Bag is abstract new Container
  with null record;
 ...
type Bag_Iterator (B : access Bag'Class)
is new Actual_Iterator (B) with record
  Bucket_Index :  Natural := 0;
  Index :  Natural := 0;
end record;
 ...
end BC.Containers.Bag;
```

Figure 5: Extract of the generic package *BC.Containers.Bags*

Due to this dependency, the number of implementation strategies for each different structure abstraction is restricted to a single one. This is a serious drawback because some of the implementations provided therein can be inefficient or inappropriate in some contexts. For instance, the hashing implementation of *Bags* (and also *Maps*) does not support ordered traversal of items and, thus, these containers are inappropriate when efficient ordered traversals are needed.

But the problem is far more severe. Not only multiple implementations are not allowed, extensions of the class hierarchy with some abstractions or changes of implementations are not always possible. For instance, it is not possible to easily change the implementation of the class *BC.Containers.Bags.Unbounded* to a tree-based strategy (for instance, when elements must be obtained in some order or when the size of the bag is not known in advance), because it is not implementation-independent, and hence it forces a concrete implementation strategy (hashing).

So, changing the hashing implementation of *BC.Containers.Bags.Unbounded* by an implementation (unbounded) with a binary search tree, requires changing the implementation of the type *Bag_Iterator* (defined in the *BC.Containers.Bags* class) and the implementation of its operations as well. Moreover, as a side-effect of this change, the implementation of the classes *BC.Containers.Bags.Bounded* and *BC.Containers.Bags.Dynamic* must be changed or modified accordingly.

Last, low level of abstraction makes the process of implementation harder. This happens when dealing with iterators. The iterator type and its operations are strongly dependent

on the concrete implementation of the underlying structure. As a consequence, for every concrete implementation of a children-class a new *Actual_Iterator* type must be defined and its operations must be overridden. This approach prevents the easy implementation of the iterator facility.

To sum up, many of the problems that the BC-Library suffers come from this kind of structural dependencies, that exist because iterators are strongly dependent on the concrete container implementation. Similar dependencies can be also found between the classes *Maps*, *Sets*, *Queues*, etc., and their respective children. We have shown in Sect. 3 that the *Shortcut* approach solves these problems because allows making iterators independent of the specific container.

5.2 Time efficiency

A last point is worth to be remarked. It is not only the lack of multiple implementations for components that damages efficiency, but also some of the iterator operations have linear cost in the worst case with respect to a certain parameter (although their amortised cost is constant). For instance, as shown in Fig. 6, the *Reset* operation could have linear cost with respect to the *Number_of_Buckets*. A similar problem occurs in the *Next* operation.

```
procedure Reset (It :  in out Bag_Iterator) is
  begin
  It.Index := 0;
  if Cardinality (It.B.all) = 0 then
    It.Bucket_Index := 0;
  else
    It.Bucket_Index := 1;
    while It.Bucket_Index<=Number_Of_Buckets(It.B.all)
    loop
      if Length (It.B.all, It.Bucket_Index) > 0 then
        It.Index := 1;
        exit;
      end if;
      It.Bucket_Index := It.Bucket_Index + 1;
    end loop;
  end if;
end Reset;
```

Figure 6: Code of reset operation of the generic package *BC.Containers.Bags*

5.3 Customising the shortcuts

Once the structural flaws of the BC Library have been identified, our goal is to apply our approach as described in sections 3 and 4, in order to get an improved version, full-compatible with the original one.

Due to this fully-compatibility goal, and also to the usage of Ada-95 instead of C++[8], there is a difference between the pattern-like proposal presented in Sect. 3 and its application to the BC Library: the *Container* interface has been enlarged by a new operation named `Shortcut_To_The_Last_Item_Added` instead of modifying the original `Add` operation to return a *shortcut*. This operation returns the shortcut associated to the last item added to a container. Then the C++ code:

```
sh = C.Add(I);
```

to add an item and obtain its shortcut, whose corresponding Ada code would be:

```
sh := Add(C,I);
```

must be really written:

```
Add(C,I);
sh := Shortcut_To_The_Last_Item_Added(C);
```

5.4 Benefits

The main results of applying the shortcut approach to the BC Library are summarized next:

- The BC Library has been improved from many points of view:
  - Step-by-step implementation. Each level of the hierarchy (i.e., step) of the new version of the library is fully implemented avoiding the dependencies of the old version.
  - Design with reuse. Due to the new internal structure (although the hierarchy class remains the same), which enhances code reuse, changes of any kind are facilitated; remarkably, adding new implementations is a straightforward process since the code for iterators is inherited (the original BC library requires implementing them over and over).
  - Implementation for reuse: generality. Unlike the original BC library, multiple implementation strategies are supported. For instance, there can be simultaneously hashing and tree implementations for bags.
  - Implementation for reuse: extensibility. The BC Library can be enlarged with as many components and implementations of components as required.
  - Functionality. In addition to the new functionalities introduced by shortcuts, that did not exist before, the new iterators are bidirectional while the old ones were not.
  - Integrity. Iterators are persistent and control of out-of-date iterators are provided. Inconsistencies arising in the original version when iterations and updates where combined are not possible now because of deleted flags, reference counts and persistence of shortcuts.
  - Time efficiency. Shortcuts and iterators are highly efficient when measured with the standard big-Oh notation [6]; accessing through them is $O(1)$ in the worst case.
- This improvement has been made in a very comfortable way; just a few changes were needed:
  - Implementation of the operations of the container base class.

[8] Being this also interesting to prove the feasibility of our approach in different programming languages.

- Implementation of shortcuts and iterators on the container base class.
- Deletion of iterators' code in the concrete class (e.g., *Bag_Iterator*).
- Instantiation of the concrete implementation of the container (e.g., implementation of *Bags.Unbounded* with *shortcut* instead of *items*). Then the core data structures and algorithms are the same without any modification at all.

- The reengineering process does not interfere with the previous behaviour of the library (both for functionality and for efficiency) and, in consequence, existing software that use this library does not need to be modified; only recompilation is needed. We remark that the operations in the original version keep their interface in the new version.

- But existing software could be modified in a methodical way (basically, changing the way of accessing to the structure) to take profit of the new version of the library. New software, of course, would be built in general using the new layout of the structure, making intensive use of shortcuts.

- The new library has been successfully tested using the test packages provided by the original BC Library without any modification.

5.5 Computational results

In this section we present the computational results corresponding to the time efficiency of some operations of the original version of the BC library and of the shortcut version. These results are presented because, although the asymptotic cost of original version is preserved by the shortcuts version, the *Add* operation suffers from a little overhead, since we need to add: first the item in the container base class, and then the corresponding shortcut in the concrete container implementation (see the corresponding code in *Implementation of the class Container* in Sect. 3, pag. 9).

So, the goal of this subsection is to determine whether this increasing cost is amortised with the cost saving of the use of shortcuts and the new version of iterators. To do this, we have tested the original version of the BC Library and its version with shortcuts on a large set of instances. We present here only some representative results[9]. In these results (see Tables 1-3 and also the appendix) we compare the execution time obtained by the original version of the library and the shortcuts version for the three representative processes: insertion, lookup and iteration in a *Bag*.

Processes.

We have measured the time of the following three processes: 1) insertion of a large amount of elements in a bag; 2) an individual access to each element previously inserted in a bag, using the item in the case of the original version and the shortcut in the version with shortcuts; 3) one single iteration over all the elements previously inserted in a bag.

[9] Complete results can be found at `http://www.lsi.upc.es/~jmarco/testing.html`.

Experimental procedure.

The experimental process has consisted in measuring the execution time (in seconds) for the three processes mentioned above in different scenarios. In order to cover the most representative scenarios we have considered the following three criteria: item size, percentage of occupation of the *Bag* and number of collisions by bucket produced by the hash function. For each scenario, different bag sizes and different number of insertions (depending of the number of occurrences of each item) have been tested.

All the test programs and the Ada-95 packages needed to generate these programs together with the shortcut version can be found at `http://www.lsi.upc.es/~jmarco/testing.html` and the original library at `http://www.pogner.demon.co.uk/components/bc`.

Instance generation.

The different instances have been randomly generated by a simple program. All the instances and the program used to generate them can be found at `http://www.lsi.upc.es/~jmarco/testing.html`.

Platform.

PC with a Pentium III processor 850 MHz with 128 Mb of RAM under Linux.

Numerical results.

We present a table for each of the 12 different scenarios coming from the combination of the following factors: type of items (String(1..8), String(1..32) and String(1..300) Ada Types), occupation of the hashing table (100% and 25%) and number of collisions (0 and 10 for each bucket). Three of these tables are shown in this section, the rest of them appear in the Appendix.

The first two columns of each table concern to the size of the bag; the first one is the number of buckets and the second one the size of each bucket (corresponding to the maximum number of collisions per bucket). The following two columns are respectively the total number of inserted items (including repetitions) and the percentage of repetitions. Finally, the three last columns are respectively: the process, the time in seconds of this process in the original version of the library, and the time in seconds of this process in the version with shortcuts.

Analysis of the results.

We can observe that the time of iteration and the time of lookup (with respect to the number of items) are independent of the scenario in the case of the shortcuts version while in the case of the original version the time increases in proportion to the size of the items and in some cases (instances are random) in inverse proportion to the percentage of occupation of the bag, and the time of lookup increases in proportion to the size of the item. In all the cases, the time of the shortcuts version is less than the time of the original version and oscillates between 1.5 and 3 times faster in the case of iteration and between 3 and 15 times faster in the case of lookup. On the other hand, such as we remark at the beginning of this section, the insertion has an overhead in the shortcuts version that oscillates between a bit more

Table 1: Computation results: String(1..8) items, 25% occupation and no collisions.

Bag size						
Buckets	Size	# inser.	% rep.	Process	Time O.V.	Time S.V.
100000	1	33333	25%	Insert	0.232727	0.626210
				Look-up	0.057436	0.010618
				Iterate	0.151795	0.052825
100000	1	100000	75%	Insert	0.754271	1.361307
				Look-up	0.057675	0.010706
				Iterate	0.152093	0.052963
10000	1	3333	25%	Insert	0.023037	0.061466
				Look-up	0.005545	0.000878
				Iterate	0.023037	0.005262
10000	1	10000	75%	Insert	0.072626	0.135230
				Look-up	0.005528	0.000882
				Iterate	0.014287	0.005185

Table 2: Computation results: String(1..32) items, 100% occupation and no collisions.

Bag size						
Buckets	Size	# inser.	% rep.	Process	Time O.V.	Time S.V.
100000	1	133333	25%	Insert	0.990184	2.518569
				Look-up	0.264771	0.041477
				Iterate	0.535682	0.223071
100000	1	400000	75%	Insert	3.333870	5.712785
				Look-up	0.264922	0.041517
				Iterate	0.534602	0.226980
10000	1	13333	25%	Insert	0.095824	0.250959
				Look-up	0.027023	0.004758
				Iterate	0.053474	0.022673
10000	1	40000	75%	Insert	0.322610	0.569477
				Look-up	0.026921	0.004733
				Iterate	0.053534	0.022269

Table 3: Computation results: String(1..300) items, 100% occupation and 10 collisions.

Bag size						
Buckets	Size	# inser.	% rep.	Process	Time O.V.	Time S.V.
100000	1	133333	25%	Insert	2.938357	4.322883
				Look-up	0.628765	0.041896
				Iterate	0.861814	0.319625
100000	1	400000	75%	Insert	8.062084	9.243259
				Look-up	0.629822	0.041978
				Iterate	0.861465	0.319821
10000	1	13333	25%	Insert	0.290843	0.413855
				Look-up	0.062429	0.004972
				Iterate	0.086572	0.032031
10000	1	40000	75%	Insert	0.795020	0.876119
				Look-up	0.062492	0.004972
				Iterate	0.086263	0.032031

than 1 and 3 times slower than in the original version. Fig 7 shows the best case (B.C. curves; Table 3 row 4) and the worst case (W.C. curves; Table 2 row 2) given the three scenarios of this section; the overhead of insertions is amortised with 2 and 8 iterations, respectively. In general, time efficiency of insertion with shortcut gets closer to the one of the original version as size of elements increases; we remark that with strings of 300 characters, there are cases where one single iteration makes our proposal better than the original one (see Table 12 in the appendix, row 4). We recall also that in the case of strings of 32 characters or more as keys for container elements, our approach saves the overall space of the program data when there are one or more references to every element of the container.

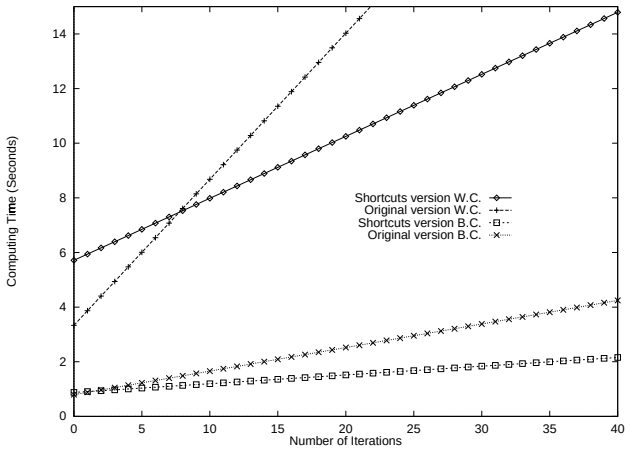


Figure 7: Comparing the original version with the shortcuts version.

6. CONCLUSIONS

We have presented in this paper a proposal trying to reconcile design and implementation in the field of container-like component libraries (CLC-libraries) with respect to three principles that should drive their functioning. Once four widespread CLC-libraries (JCF, STL, BC and LEDA) have been examined, we have concluded that there is a gap between design and implementation. This is the motivation of our work.

Next we have presented an approach for bridging this gap around the idea of shortcut, that is, efficient and safe access by position. We have defined a framework around a design pattern that makes possible the definition of this kind of access, together with the access by iterators, fulfilling the three implementation principles: step-by-step implementation, implementation with reuse and implementation for reuse. We have applied our approach to an existing CLC-library and we have then proven its feasibility not only in building new CLC-libraries, but also in reengineering existing ones, just by customising the approach mainly from a syntactical point of view.

In addition of their utility for satisfying the three implementation principles, remarkably supporting full code reuse, the use of shortcuts impacts positively on quality of CLC-libraries as follows:

- Suitability. In addition to the originally intended functionality of the CLC-library, other operations for accessing the containers therein can be added in a well-defined and uniform way, without interfering with previously existing operations.
- Adaptability. The resulting CLC-library does not make any assumption about its context of use. In fact, the existence of shortcuts increase the adaptability degree of CLC-libraries, because it allows their integration in contexts with strong efficiency constraints, both for the access by position itself and for the absence of restrictions on the candidate implementations of containers (hashing, trees, etc.).
- Changeability. Due to the internal structure of the resulting CLC-library, addition of new components (or addition of new implementations for existing ones) not only is allowed but also may take profit of existing shortcuts without having to implement them again.
- Integrity. Access to the container by means of out-of-date shortcuts is detected and avoided. Furthermore, as far as operation using shortcuts do not interfere with the other operations of the container, the internal data structure is kept safe.
- Applicability. The approach is not only useful for building new libraries but also for reengineering existing ones, obtaining a new version fully-compatible with the original one; thus, the new version may substitute the original one and the running applications built on top of the library do not require any kind of actualisation.
- Time efficiency. Shortcuts allow highly efficient access to elements without interfering in a significant manner with the efficiency of the other operations.
- Formal basis. The definition of shortcuts is highly abstract and well-suited for being formally specified, as we have done in a previous version by means of an equational specification [12]).

From the study carried out in Sect. 2, we may conclude that even the most widespread CLC-libraries fail in implementing access by position while satisfying not only the implementation principles but also these quality criteria altogether up to an acceptable level.

Concerning the potential drawbacks of our approach, we think they are restricted to the need of spending some additional space for the underlying data structures representing the containers, and also to the arising of a small penalty on the time efficiency of insertion operations. Nevertheless, both overheads may be amortised by later savings; in the case of space, external references are generally smaller with shortcuts than without them; in the case of time, traversals and individual lookups are faster with shortcuts than without them. We have computed the formulae that fix the space of shortcut addition, and we have also developed an

exhaustive analysis of time efficiency by means of representative scenarios in the case study mentioned above; then, we have deduced the relationships that must hold in order to save space and time with shortcut addition.

7. REFERENCES

[1] S. Tucker Taft and R.A. Duff (Eds.). *Ada 95 Reference Manual.* Lecture Notes in Computer Science 1246, Springer-Verlag, 1995.

[2] K. Arnold, J. Gosling and D. Holmes. *The Java Programming Language.* Addison-Wesley, 3rd edition, 2000.

[3] Accredited Standards Committe X3 (ANSI), Information Processing Systems, *Working Paper for Draft Proposed International Standard for Information Systems–Programming Language C++.* Doc No.X3J16/95-0185, WG21/N0785.

[4] F. Bachman et al. *Technical Concepts of Component-Based Software Engineering.* Informe del Software Engineering Institute de la Carnegie Mellon University, CMU/SEI-2000-TR-008, 2000.

[5] G. Booch. *Software Components with Ada.* The Benjamin/Cummings Publishing Company, 2nd edition, 1987.

[6] G. Brassard and P. Bratley. *Fundamentals of Algorithmics.* Prentice Hall, 1996.

[7] R. Breu. Algebraic Specification Techniques in Object Oriented Programming Environments. Lecture Notes in Computer Science 562, Springer-Verlag, 1991.

[8] G. Booch and M. Vilot. The Design of the C++ Booch Components. In *Proceedings of Conference on Object Oriented-Programming: Systems, Languages and Applications (OOPSLA)*, volume 25 of *SIGPLAN Notices*, pages 1-11. ACM, 1990.

[9] G. Booch, D.G. Weller and S. Wright. The Booch Library for Ada 95 (version 1999). Available at http://www.pogner.demon.co.uk/components/bc.

[10] L.P. Deutsch. Design reuse and frameworks in the smalltalk-80 system. *Software Reusability, Volume II. Applications and Experience*, ACM, 1989.

[11] D.R. Edelson. Smart pointers: They're smart, but they're not pointers. In *Proceedings of the 1992 USENIX C++ Conference*, pages 1-19. USENIX Association, 1990.

[12] X. Franch and J. Marco. Adding Alternative Access Paths to Abstract Data Types. In *Challenges of Information Technology Management in the 21st Century (IRMA'2000)*, pages 283-287. Idea Group Publishing, 2000.

[13] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1996.

[14] R.E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22-35, June/July 1988.

[15] G. Kiczales and J. Lamping. Issues in the Design and Specification of Class Libraries. In *Proceedings of Conference on Object Oriented-Programming: Systems, Languages and Applications (OOPSLA'92),SIGPLAN Notices*, pages 435-451. ACM, 1992.

[16] B. Meyer. *Reusable Software: the base object-oriented component libraries.* Prentice Hall, 1994.

[17] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 2nd edition, 1997.

[18] M. McIlroy Mass Produced Software Engineering In *Software Engineering Concepts and Techniques.* NATO Conference on System Sciences, 1969.

[19] J. Marco and X. Franch. Reengineering the Booch Component Library In *Reliable Software Technologies Ada-Europe 2000*, volume 1845 of *Lecture Notes in Computer Science*, pages 96-111. Springer-Verlag, 2000.

[20] J. Marco and X. Franch. Bridging the Gap Between Design and Implementation of Component Libraries (extended version). Technical Report, Departament de Llenguatges i Sistemes Informàtics. Universitat Politècnica de Catalunya, 2000.

[21] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing.* Cambridge University Press, 1999.

[22] D.R. Musser and A. Saini. *STL Tutorial and Reference Guide.* Addison-Wesley, 1996.

[23] O. Nierstraz and D. Tsichritzis. *Object-Oriented Software Composition.* Prentice Hall, 1996.

[24] C. Szyperski. *Component Software – Beyond Object-Oriented Programming.* Addison-Wesley, 1998.

[25] R. Wilhem and D. Maurer. *Compiler Design.* Addison-Wesley, 1995.

APPENDIX

Table 4: Computation results: String(1..8) items, 100% occupation and no collisions.

Bag size						
Buckets	Size	# inser.	% rep.	Process	Time O.V.	Time S.V.
100000	1	133333	25%	Insert	0.931205	2.507402
				Look-up	0.212484	0.041629
				Iterate	0.473699	0.210886
100000	1	400000	75%	Insert	3.026749	5.462093
				Look-up	0.213501	0.041476
				Iterate	0.476947	0.210438
10000	1	13333	25%	Insert	0.091898	0.246780
				Look-up	0.021662	0.004625
				Iterate	0.044849	0.021074
10000	1	40000	75%	Insert	0.293019	0.541389
				Look-up	0.021733	0.004625
				Iterate	0.044821	0.021218

Table 5: Computation results: String(1..8) items, 100% occupation and 10 collisions.

Bag size						
Buckets	Size	# inser.	% rep.	Process	Time O.V.	Time S.V.
10000	10	133333	25%	Insert	1.041547	3.962840
				Look-up	0.225440	0.041350
				Iterate	0.424440	0.219884
10000	10	400000	75%	Insert	3.154813	7.645176
				Look-up	0.225552	0.041374
				Iterate	0.428446	0.219860
1000	10	13333	25%	Insert	0.098534	0.377556
				Look-up	0.022852	0.004598
				Iterate	0.098534	0.021932
1000	10	40000	75%	Insert	0.309070	0.720568
				Look-up	0.022999	0.004593
				Iterate	0.042413	0.022009

Table 6: Computation results: String(1..8) items, 25% occupation and 10 collisions.

Bag size						
Buckets	Size	# inser.	% rep.	Process	Time O.V.	Time S.V.
10000	10	33333	25%	Insert	0.242589	0.733268
				Look-up	0.055689	0.010656
				Iterate	0.108487	0.054909
10000	10	100000	75%	Insert	0.754532	1.521262
				Look-up	0.055604	0.010627
				Iterate	0.108377	0.054839
1000	10	3333	25%	Insert	0.023347	0.070583
				Look-up	0.005307	0.000991
				Iterate	0.010510	0.005486
1000	10	10000	75%	Insert	0.073936	0.144295
				Look-up	0.005357	0.001017
				Iterate	0.010508	0.005456

Table 7: Computation results: String(1..32) items, 25% occupation and no collisions.

Bag size						
Buckets	Size	# inser.	% rep.	Process	Time O.V.	Time S.V.
100000	1	33333	25%	Insert	0.246659	0.626989
				Look-up	0.070232	0.010603
				Iterate	0.170482	0.055686
100000	1	100000	75%	Insert	0.830515	01.420188
				Look-up	0.070054	0.010682
				Iterate	0.170094	0.056357
10000	1	3333	25%	Insert	0.023986	0.062630
				Look-up	0.006943	0.001024
				Iterate	0.016945	0.005724
10000	1	10000	75%	Insert	0.079533	0.140674
				Look-up	0.006920	0.001042
				Iterate	0.016896	0.005486

Table 8: Computation results: String(1..32) items, 100% occupation and 10 collisions.

Bag size						
Buckets	Size	# inser.	% rep.	Process	Time O.V.	Time S.V.
10000	10	133333	25%	Insert	1.188675	3.987754
				Look-up	0.288713	0.042438
				Iterate	0.409925	0.226806
10000	10	400000	75%	Insert	3.678540	8.028995
				Look-up	0.288720	0.042636
				Iterate	0.491895	0.226806
1000	10	13333	25%	Insert	0.108513	0.383991
				Look-up	0.027920	0.004725
				Iterate	0.048574	0.022615
1000	10	40000	75%	Insert	0.337034	0.7505710
				Look-up	0.027947	0.004700
				Iterate	0.048443	0.022441

Table 9: Computation results: String(1..32) items, 25% occupation and 10 collisions.

Bag size						
Buckets	Size	# inser.	% rep.	Process	Time O.V.	Time S.V.
10000	10	33333	25%	Insert	0.256243	0.740662
				Look-up	0.069334	0.011048
				Iterate	0.123223	0.056315
10000	10	100000	75%	Insert	0.836068	1.614531
				Look-up	0.069303	0.010822
				Iterate	0.123120	0.055959
1000	10	3333	25%	Insert	0.023808	0.070950
				Look-up	0.006448	0.001080
				Iterate	0.011967	0.005752
1000	10	10000	75%	Insert	0.077318	0.149599
				Look-up	0.006520	0.001074
				Iterate	0.012002	0.005506

Table 10: Computation results: String(1..300) items, 100% occupation and no collisions.

Bag size						
Buckets	Size	# inser.	% rep.	Process	Time O.V.	Time S.V.
100000	1	133333	25%	Insert	1.254917	2.873730
				Look-up	0.608860	0.041993
				Iterate	0.871690	0.320561
100000	1	400000	75%	Insert	4.868695	6.970321
				Look-up	0.606949	0.041990
				Iterate	0.871956	0.319978
10000	1	13333	25%	Insert	0.164564	0.281967
				Look-up	0.059941	0.004785
				Iterate	0.087135	0.032379
10000	1	40000	75%	Insert	0.602049	0.690067
				Look-up	0.059942	0.004789
				Iterate	0.086986	0.032188

Table 11: Computation results: String(1..300) items, 25% occupation and no collisions.

Bag size						
Buckets	Size	# inser.	% rep.	Process	Time O.V.	Time S.V.
100000	1	33333	25%	Insert	0.423690	0.701949
				Look-up	0.155360	0.011002
				Iterate	0.255654	0.080232
100000	1	100000	75%	Insert	1.559128	1.721003
				Look-up	0.155308	0.011084
				Iterate	0.255464	0.080005
10000	1	3333	25%	Insert	0.041132	0.070021
				Look-up	0.015350	0.000824
				Iterate	0.025648	0.008294
10000	1	10000	75%	Insert	0.149465	0.169829
				Look-up	0.015252	0.000824
				Iterate	0.025648	0.008044

Table 12: Computation results: String(1..300) items, 25% occupation and 10 collisions.

Bag size						
Buckets	Size	# inser.	% rep.	Process	Time O.V.	Time S.V.
10000	10	33333	25%	Insert	0.546532	0.815297
				Look-up	0.155545	0.011137
				Iterate	0.217140	0.080164
10000	10	100000	75%	Insert	1.785882	1.897252
				Look-up	0.155398	0.011094
				Iterate	0.217235	0.079796
1000	10	3333	25%	Insert	0.054132	0.078701
				Look-up	0.015208	0.000813
				Iterate	0.021671	0.008199
1000	10	10000	75%	Insert	0.179306	0.181203
				Look-up	0.015182	0.000812
				Iterate	0.021515	0.007990